Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 255–264
http://www.floc-conference.org/ICLP-home.html

CONSTRAINT ANSWER SET PROGRAMMING SYSTEMS

CHRISTIAN DRESCHER [1]

[1] Institute of Information Systems,
Vienna University of Technology, Favoritenstraße 9-11, A-1040 Vienna, Austria
E-mail address: christian.drescher@student.tuwien.ac.at

Abstract. We present an integration of answer set programming and constraint processing as an interesting approach to constraint logic programming. Although our research is in a very early stage, we motivate constraint answer set programming and report on related work, our research objectives, preliminary results we achieved, and future work.

1. Introduction

Constraint satisfaction problems (CSP) are combinatorial problems defined as a set of variables whose value must satisfy a number of limitations, and are subject to intense research. Problems that has been successfully modelled as a CSP stem from a variety of areas, for example, artificial intelligence, operations research, electrical engineering and telecommunications.

There are several approaches to representing and solving constraint satisfaction problems: constraint programming (CP; [Ros06]), answer set programming (ASP; [Bar03]), propositional satisfiability checking (SAT; [Bie09]), its extension to satisfiability modulo theories (SMT; [Nie06]), and many more. Each has its particular strengths: for example, CP systems support global constraints, ASP systems permit recursive definitions and offer default negation, whilst SAT solvers often exploit very efficient implementations. In many applications it would often be helpful to exploit the strengths of multiple approaches. Consider the problem of timetabling at an university (cf. [Jär09]). To model the problem, we need to express the mutual exclusion of events (for instance, we cannot place two events in the same room at the same time). A straightforward representation of such constraint with clauses and rules uses quadratic space. In contrast, global constraints such as *all-different* typically supported by CP systems can give a much more concise encoding. On the other hand, there are features which are hard to describe in traditional constraint programming, like the temporary unavailability of a particular room. However, this is easy to represent with non-monotonic rules such as those used in ASP. Such rules also provide a flexible mechanism for defining new relations on the basis of existing ones. This makes answer set programming an attractive approach to declarative problem solving. Indeed, ASP has been shown as the computational embodiment of non-monotonic reasoning (NMR; [Rei87]), adequate for common-sense reasoning and modelling of dynamic and incomplete knowledge.

As a primary candidate for an effective tool for knowledge representation and reasoning, ASP combines an expressive language with high-performance solving capacities. Largely

Key words and phrases: answer set programming, constraint logic programming, constraint processing.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.255

based on SAT technology, modern ASP solvers offer an efficiency and scalability which in practice remain largely unmatched to date [Geb07a], able to encode all search problems within the first three levels of the polynomial hierarchy [Dre08]. Particularly of relevance here is the fact that clause learning is known to be more general and potentially more powerful than traditional learning in constraint solvers [Kat05]. Unlike SAT, however, ASP offers a uniform modelling language admitting variables. In fact, grounding non-propositional specifications is addressed in SAT anew for each application while ASP centralized this task in its grounders [Syr, Geb07c]. Answer set programming has been shown to be useful in numerous application scenarios, like bioinformatics [Bar04], crypto analysis [Aie01], configuration [Soi99], database integration [Leo05], diagnosis [Eit99], hardware design [Erd], model checking [Hel03], planing [Lif02], preference reasoning [Bre96], semantic web [Eit08], and – as a highlight among these applications – the high-level control of the space shuttle [Nog01].

However, as some CSP are more naturally modelled by using non-propositional constructs, like resources or functions over finite domains, in particular global constraints, the need to handle constraints beyond pure ASP is increasing. This naturally leads to the combination of ASP with constraint processing [Dec03] techniques, and is target of our research activity. A key contribution of our work is a novel approach to constraint logic programming (CLP; [Jaf94]) centred around ASP as both a declarative specification language and an efficient reasoning engine, enhanced with specialised propagators sufficient to solve interesting constraint satisfaction problems.

This research summary is organized as follows. We start by giving the necessary background and an overview of the existing literature. In turn, we formulate our objectives in Section 3. Section 4 gives a brief overview of the current status of our research and Section 5 presents some preliminary results. The last part summarises open questions which are target to our future work.

2. Background

2.1. Answer Set Programming

A *(normal) logic program* Π over a set of primitive propositions $\mathcal{A}$ is a finite set of rules of the form

$$a_0 \leftarrow a_1, \dots, a_m, \mathit{not}\ a_{m+1}, \dots, \mathit{not}\ a_n \tag{2.1}$$

where $0 \leq m \leq n$ and $a_i \in \mathcal{A}$ is an *atoms* for $0 \leq i \leq n$. A *literal* $\hat{a}$ is an atom a or its default negation *not* a. For a rule r, let $\mathit{head}(r) = a_0$ be the *head* of r and $\mathit{body}(r) = \{a_1, \dots, a_m, \mathit{not}\ a_{m+1}, \dots, \mathit{not}\ a_n\}$ the *body* of r. Furthermore, define $\mathit{body}(r)^+ = \{a_1, \dots, a_m\}$ and $\mathit{body}(r)^- = \{a_{m+1}, \dots, a_n\}$. The set of atoms occurring in a logic program Π is denoted by $\mathit{atom}(\Pi)$, and the set of bodies in Π is $\mathit{body}(\Pi) = \{\mathit{body}(r) \mid r \in \Pi\}$. For regrouping bodies sharing the same head a, define $\mathit{body}(a) = \{\mathit{body}(r) \mid r \in \Pi,\ \mathit{head}(r) = a\}$.

The semantics of a program is given by its answer sets. A set $X \subseteq \mathcal{A}$ is an *answer set* of a logic program Π over $\mathcal{A}$, if X is the $\subseteq$-minimal model of the *reduct* [Gel88]

$$\Pi^X = \{\mathit{head}(r) \leftarrow \mathit{body}(r)^+ \mid r \in \Pi,\ \mathit{body}(r)^- \cap X = \emptyset\}.$$

A rule r of the form (2.1) can be seen as a constraint on the answer sets of a program, stating that if $a_1, \ldots, a_m$ are in the answer set and none of $a_{m+1}, \ldots, a_n$ are included, then a_0 must be in the set. The answer sets are the key objects of interest in this paradigm and, hence, the task of ASP systems is to compute answer sets for programs. Such a system differs substantially from traditional logic programming systems, such as Prolog, which are goal-directed backward chaining query evaluation systems.

The semantics of important extensions to normal logic programs, such as choice rules, integrity and cardinality constraints, is given through program transformations that introduce additional propositions (cf. [Sim02]). A *choice rule* allows for the non-deterministic choice over atoms in $\{h_1, \ldots, h_k\}$ and has the following form:

$$\{h_1, \ldots, h_k\} \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n \tag{2.2}$$

An *integrity constraint*

$$\leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n \tag{2.3}$$

is a short hand for a rule with an unsatisfiable head, and thus forbids its body to be satisfied in any answer set. A *cardinality constraint*

$$\leftarrow k\{a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n\} \tag{2.4}$$

is interpreted as no k literals of the set $\{a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n\}$ are included in an answer set. [Sim02] provides a transformation that needs just $\mathcal{O}(nk)$ rules. Alternatively, modern ASP solvers also incorporate specialised propagators for cardinality constraints that run in $\mathcal{O}(n)$.

Although the answer set semantics are propositional, atoms in $\mathcal{A}$ and can be constructed from a first-order signature $\Sigma_{\mathcal{A}} = (\mathcal{F}_{\mathcal{A}}, \mathcal{V}_{\mathcal{A}}, \mathcal{P}_{\mathcal{A}})$, where $\mathcal{F}_{\mathcal{A}}$ is a set of function symbols (including constant symbols), $\mathcal{V}_{\mathcal{A}}$ is a denumerable collection of (first-order) variables, and $\mathcal{P}_{\mathcal{A}}$ is a set of predicate symbols. The logic program over $\mathcal{A}$ is then obtained by a grounding process, systematically substituting all occurrences of variables $\mathcal{V}_{\mathcal{A}}$ by terms in $\mathcal{T}(\mathcal{F}_{\mathcal{A}})$, where $\mathcal{T}(\mathcal{F}_{\mathcal{A}})$ denotes the set of all ground terms over $\mathcal{F}_{\mathcal{A}}$. Atoms in $\mathcal{A}$ are formed from predicate symbols $\mathcal{P}_{\mathcal{A}}$ and terms in $\mathcal{T}(\mathcal{F}_{\mathcal{A}})$.

ASP systems usually use a *generate-and-test* [Bar03] technique to model a problem, by producing the space of solution candidates in the *generate* component and defining rules that filter invalid solutions in the *test* component. Typically, solutions are computed by applying *conflict-driven nogood learning* (CDNL; [Geb07b]). This combines search and propagation by recursively assigning the value of a proposition and using *unit-propagation* to determine logical consequences [Mit05].

2.2. Constraint Satisfaction Problem

The classic definition of a constraint satisfaction problem is as follows (cf. [Ros06]). A CSP is a triple (V, D, C) where V is set *variables* $V = \{v_1, \ldots, v_n\}$, D is an set of (finite) *domains* $D = \{D_1, \ldots, D_n\}$ such that each variable v_i has an associated domain $dom(v_i) = D_i$, and C is a set of *constraints*. A constraint c is a pair (R_S, S) where R_S is a k-ary *relation* on the variables in $S \subseteq V^k$, called the *scope* of c. In other words, R_S is a subset of the Cartesian product of the domains of the variables in S. To access the relation and the scope of c define $range(c) = R_S$ and $scope(c) = S$. For a *(constraint variable) assignment* $A : V \rightarrow \bigcup_{v \in V} dom(v)$ and a constraint $c = (R_S, S)$ with $S = (v_1, \ldots, v_k)$, define $A(S) = (A(v_1), \dots, A(v_k))$, and call c *satisfied* if $A(S) \in \mathit{range}(c)$. Given this, define the set of constraints satisfied by A as $\mathit{sat}_C(A) = \{c \mid A(\mathit{scope}(c)) \in \mathit{range}(c),\ c \in C\}$.

A binary constraint c has $|\mathit{scope}(c)| = 2$. For example, $v_1 \neq v_2$ ensures that v_1 and v_2 take different values. A global (or n-ary) constraint c has parametrized scope. For example, the *all-different* constraint ensures that a set of variables, $\{v_1, \dots, v_n\}$ take all different values. This can be decomposed into $O(n^2)$ binary constraints, $v_i \neq v_j$ for $i < j$. However, such decomposition can hinder inference [Wal00]. An assignment A is a *solution* for a CSP iff it satisfies all constraints in C.

2.3. Constraint Programming

Constraint programming is a natural paradigm for solving constraint satisfaction problems. CP systems usually use a *constrain-and-generate* technique in which an initial deterministic phase assigns a domain to each of the constraint variables and imposes a number of constraints, then a non-deterministic phase generates and explores the solution space. Various heuristics affecting, for instance, the variable selection criteria and the ordering of the attempted values, can be used to guide the search. Each time a variable is assigned a value, a deterministic propagation stage is executed, pruning the set of values to be attempted for the other variables, i.e., enforcing a certain type of local consistency.

A binary constraint c is called *arc consistent* iff when a variable $v_1 \in \mathit{scope}(c)$ is assigned any value $d_1 \in \mathit{dom}(v_1)$, there exists a consistent value $d_2 \in \mathit{dom}(v_2)$ for the other variable v_2. An n-ary constraint c is *hyper-arc consistent* or *domain consistent* iff when a variable $v_i \in \mathit{scope}(c)$ is assigned any value $d_i \in \mathit{dom}(v_i)$, there exist compatible values in the domains of all the other variables $d_j \in \mathit{dom}(v_j)$ for all $1 \leq j \leq n,\ j \neq i$ such that $(d_1, \dots, d_n) \in \mathit{range}(c)$.

The concepts of bound and range consistency are defined for constraints on ordered intervals. Let $\mathit{min}(D_i)$ and $\mathit{max}(D_i)$ be the minimum value and maximum value of the domain D_i. A constraint c is *bound consistent* iff when a variable v_i is assigned $d_i \in \{\mathit{min}(\mathit{dom}(v_i)), \mathit{max}(\mathit{dom}(v_i))\}$ (i.e. the minimum or maximum value in its domain), there exist compatible values between the minimum and maximum domain value for all the other variables in the scope of the constraint. Such an assignment is called a *bound support.* A constraint is *range consistent* iff when a variable is assigned any value in its domain, there exists a bound support. Notice that range consistency is in between domain and bound consistency.

2.4. Constraint Logic Programming

Constraint logic programming is a programming paradigm that naturally merges traditional constraint programming and logic programming. The goal is to bring advantages of logic programming based knowledge representation techniques to constraint programming.

Formally, a *constraint logic program* Π is defined as logic programs over an extended alphabet distinguishing regular and constraint atoms, denoted by $\mathcal{A}$ and $\mathcal{C}$, respectively, such that $\mathit{head}(r) \in \mathcal{A}$ for each $r \in \Pi$. Observe that a constraint logic program without constraints is in fact a (normal) logic program. Constraint atoms are identified with constraints via a function $\gamma : \mathcal{C} \to C$. For sets of constraints, define $\gamma(C') = \{\gamma(c) \mid c \in C'\}$ for $C' \subseteq C$. Similar to (normal) logic programs, the atoms in $\mathcal{A}$ and $\mathcal{C}$ can be constructed from a multi-sorted, first-order signature $\Sigma = (\mathcal{F}_\mathcal{A} \cup \mathcal{F}_\mathcal{C}, \mathcal{V}_\mathcal{A} \cup \mathcal{V}_\mathcal{C}, \mathcal{P}_\mathcal{A} \cup \mathcal{P}_\mathcal{C})$, where $\mathcal{F}_\mathcal{A} \cup \mathcal{F}_\mathcal{C}$ is a finite set of function symbols (including constant symbols), $\mathcal{V}_\mathcal{A}$ is a denumerable collection of regular variable symbols, $\mathcal{V}_\mathcal{C} \subseteq \mathcal{T}(\mathcal{F}_\mathcal{A})$ is a set of constraint variable symbols, and $\mathcal{P}_\mathcal{A} \cup \mathcal{P}_\mathcal{C}$ is a finite set of predicate symbols, where $\mathcal{P}_\mathcal{A}$ and $\mathcal{P}_\mathcal{C}$ are disjoint. While the atoms in $\mathcal{A}$ are formed as discussed before, the ones in $\mathcal{C}$ are constructed from predicate symbols $\mathcal{P}_\mathcal{C}$ and $(\mathcal{F}_\mathcal{C}, \mathcal{V}_\mathcal{C})$-terms. This definition follows Gebser et. al. [Geb09c] and tolerates occurrences of similar ground terms in atoms of both $\mathcal{A}$ and $\mathcal{C}$.

An integration of constraint and logic programming has been studied mainly from the point of view of extending Prolog implementations by allowing, e.g., constraints on finite domains in the rules and by integrating the necessary constraint solvers into the logic programming system. Although a Prolog-based CLP approach follows the constrain-and-generate technique from constraint programming systems, it has many advantages, including recursive definitions.

However, this significantly differs from our approach where the rules have a declarative semantics and can be understood themselves as constraints on solutions for the program.

2.5. Constraint Answer Set Programming

We extend the answer set semantics to constraint logic programs and define the *constraint reduct* as

$$\begin{aligned}\Pi^A = \{&\mathit{head}(r) \leftarrow \mathit{body}(r)|_\mathcal{A} \mid r \in \Pi,\\ &\gamma(\mathit{body}(r)^+|_\mathcal{C}) \subseteq \mathit{sat}_C(A),\ \gamma(\mathit{body}(r)^-|_\mathcal{C}) \cap \mathit{sat}_C(A) = \emptyset\}.\end{aligned}$$

Then, a set $X \subseteq \mathcal{A}$ is a *constraint answer set* of Π with respect to A, if X is an answer set of Π^A. An open question which is target to intensive research is how to efficiently incorporate answer set programming engines and constraint processing, i.e., how to generate assignments and enforce satisfaction (or violation, respectively) of constraints in $\gamma(\mathcal{C})$. We identified three different approaches: (1) translation-based techniques, (2) integration of constraint solvers, and (3) usage of additional propagators, such as aggregates.

Translation-based Techniques. Generally, in a translation-based approach all parts of the model are mapped into a single constraint language for which highly efficient off-the-shelf solvers are available. Previous work has mostly focussed on the translation of specific types of constraints to SAT. For example, pseudo-Boolean constraints (linear constraints over Boolean variables), including the special case of Boolean cardinality constraints, have been Booleanised such that a SAT solver can compete with the best existing native pseudo-Boolean solvers [Eén06, Sin05, Bai03, Bai06, Bai09]. Integer linear constraints have also been translated to SAT by transforming all constraints into primitive comparisons, of the form $v \leq c$, and encoding each of these by a different Boolean variable for each integer variable v and integer value c [Tam06].

Although efficient, these results have a number of limitations. First, the types of constraints dealt with are limited. Second, the techniques proposed are not necessarily compatible, thus making the translation of a heterogeneous constraint model difficult in both practice and theory. The latter is faced in [Hua08] presenting translation techniques to SAT at language level by systematically Booleanising a general constraint language, rather than specialised constraint types. However, this comes with the price of weaker encodings in terms of propagation power and loss of explicit domain knowledge and structure. It remains a difficult task to define universal SAT encodings that are both compact and enforce a strong type of consistency on the original model.

ASP is put forward as a constraint programming paradigm in [Nie99a], also showing that answer set programming embeds SAT but provides a more expressive framework from a knowledge representation point of view. An empirical comparison of the performance of ASP and CLP systems on solving combinatorial problems in [Dov09] proves ASP encodings to be more compact, more declarative, and highly competitive. However, techniques for translating constraint variables and constraint propagation algorithms to ASP received few attention in our context. A first study on introducing high-level statements for multi-valued propositions into the language of ASP was conducted in [Geb09a]. As we shall see, a translational approach to constraint answer set solving [Dre10b] offers an efficient way to seamlessly combine the propagators of all constraints, through the unit-propagation of an ASP solver. In particular, queueing of propagators becomes irrelevant as all constraints are always propagated at once. Another major strength is that the unified conflict resolution framework can exploit constraint interdependencies, which may lead to faster propagation between constraints.

Hybrid Approach. In a hybrid system, theory-specific solvers interact in order to compute solutions to the whole constraint model, similar to satisfiability modulo theories. Hence, the key idea of an integrative approach is to incorporate constraint predicates into propositional formulas, and extending an ASP solver's decision engine for a more high-level proof procedure. This becomes increasingly attractive in constraint answer set programming when the variables in a constraint model have significantly large domains, and therefore, computing the ground instantiation has huge memory requirements [Pal09]. Related work was conducted in [Geb09c, Bas, Mel08b, Mel08a]. While Gelfond et. al. [Bas, Mel08b, Mel08a] view ASP and CP solvers as blackboxes, Gebser et. al. [Geb09c] embed a CP solver into an ASP solver adding support for advanced backjumping and conflict-driven learning techniques. However, the computational impact compared to traditional CP is limited, first, because their methods lack support for global constraints, and second, the communication between the ASP and CP solvers with respect to learning constraint interdependencies is restricted. Balduccini [Bal09] added support for global constraints but sees constraint answer set programming largely as a CSP specification language. In particular, his approach does not allow constraint literals in the body of a rule, which does not coincide with our general notion of constraint logic programming.

Formulation of Additional Propagators. Little attention is paid to constraint answer set programming through decomposition to ASP with usage of additional propagators, such as aggregates. Aggregations and other forms of set constructions have been shown to be useful extensions to ASP [Del]. In fact, a lack of aggregation capabilities may lead to an exponential growth in the number of rules required to model a CSP [Bar03]. Therefore, it is common to most ASP solvers to incorporate specialised algorithms, for instance, the treatment of cardinality constraints (2.4), and their generalisation to weight constraints [Nie99b]. Work on a generic framework which provides an elegant treatment of such extensions was conducted in [Elk] where external constraint propagators are employed for their handling. However, it does not carry over to modern ASP solving technology based on conflict-driven learning. A first comprehensive approach to integrating specialised algorithms for weight constraint rules into CDNL is presented in [Geb09b].

3. Research Objective

We want to put forward constraint answer set programming as a novel approach to constraint (logic) programming. Therefore, we (1) investigate efficient encodings of propagation algorithms in answer set programming, (2) study the integration of techniques from constraint processing into answer set programming engines, and (3) define a modelling language for constraint logic programming under answer set semantics, that can be accepted by the community. Furthermore, we (4) want to implement our techniques in state-of-the-art systems.

4. State of the Research

Our research is in a very early stage. In a Master's project we introduced a novel, translation-based approach to constraint answer set solving [Dre10b] that allows for learning constraint interdependencies to improve propagation between constraints. As part of a Master's thesis we also started an investigation of symmetry-breaking in the context of answer set programming to eliminate symmetric parts of the search space and, thereby, simplify the solution process [Dre10a].

5. Preliminary Results

In our translational approach to constraint answer set solving, a constraint logic program is compiled into a logic program by adding an ASP decomposition of all constraints comprised in the constraint logic program. The constraint answer sets can then be obtained by applying the same algorithms as for calculating answer sets, e.g. CDNL. Since all variables will be shared between constraints, nogood learning techniques as in CDNL exploit constraint interdependencies. This can improve propagation between constraints. We identify a number of choices of how to decompose constraints on multi-valued propositions, e.g. constraint variables, in answer set programming. Namely, we propose a *direct*, *support*, *range*, and *bound* representation of constraints [Dre10b] each generically encoding a propagation algorithm in ASP (using rule types 2.1–2.4) that achieves, e.g., arc, range and bound consistency on the original constraint (or its binary decomposition, respectively), using unit-propagation. In particular, we present the following results:

Theorem 5.1. *Enforcing arc consistency on the binary decomposition of the original constraint prunes more values from the variables domain than unit-propagation on its direct encoding.*

Theorem 5.2. *Unit-propagation on the support encoding enforces arc consistency on the binary decomposition of the original constraint.*

Theorem 5.3. *Unit-propagation on the range encoding enforces range consistency on the original constraint.*

Theorem 5.4. *Unit-propagation on the bound encoding enforces bound consistency on the original constraint.*

We illustrate our approach on an encoding of the global all-different constraint enforcing, e.g., bound consistency by pruning Hall intervals [Lec96]. Surprisingly, a very simple decomposition into ASP can simulate a complex propagation algorithm like from

Leconte's [Lec96] with a similar overall complexity of reasoning. Our techniques were formulated as preprocessing and can be applied to any ASP system without changing its source code, which allows for programmers to select the solvers that best fit their needs. We have empirically evaluated their performance on benchmarks from CSP and found them outperforming integrated constraint answer set programming systems as well as pure CP solvers.

However, many CSP exhibit symmetries which can frustrate a search algorithm to fruitlessly explore independent symmetric subspaces. We have investigated symmetry-breaking in the context of answer set programming [Dre10a]. In particular, we propose a reduction from symmetry detection of disjunctive logic programs to the automorphisms of a coloured digraph. Our techniques are formulated as a completely automated flow that (1) starts with a logic program, (2) detects all of its permutational symmetries, (3) represents all symmetries implicitly and always with exponential compression, (4) adds symmetry-breaking constraints that do not affect the existence of answer sets. We have empirically evaluated symmetry-breaking on difficult CSP and got promising results. In many cases, symmetry-breaking lead to significant pruning of the search space and yield solutions to problems which are otherwise intractable. We also observe a significant compression of the solution space which makes symmetry-breaking attractive whenever all solutions have to be post-processed.

6. Future Work

Regarding symmetry-breaking answer set solving, it is often reasonable to assume that the symmetries for a problem are known. For particular symmetries, there are more efficient breaking methods, for instance, the global value precedence constraint (cf. [Wal06]).

Therefore, future work concerns, but is not limited to, the integration of further constraints useful in constraint answer set programming. In particular, we are interested in decompositions of constraints using the full range of propagators available in state-of-the-art ASP systems, and if necessary, extending ASP solving by further useful algorithms that make constraint answer set programming an efficient approach to constraint logic programming.

Acknowledgements

The author wishes to acknowledge the continuing support of Thomas Eiter, Torsten Schaub, and Toby Walsh.

References

[Aie01] L. Aiello and F. Massacci. Verifying security protocols as planning in logic programming. *ACM Transactions on Computational Logic*, 2(4):542–580, 2001.

[Bai03] O. Bailleux and Y. Boufkhad. Efficient CNF encoding of boolean cardinality constraints. In *Proceedings of CP'03*, pp. 108–122. Springer, 2003.

[Bai06] O. Bailleux, Y. Boufkhad, and O. Roussel. A translation of pseudo boolean constraints to SAT. *Journal of Satisfiability*, 2(1-4):191–200, 2006.

[Bai09] O. Bailleux, Y. Boufkhad, and O. Roussel. New encodings of pseudo-boolean constraints into CNF. In *Proceedings of SAT'09*, pp. 181–194. Springer, 2009.

[Bal09] M. Balduccini. CR-prolog as a specification language for constraint satisfaction problems. In *Proceedings of LPNMR'09*, pp. 402–408. Springer, 2009.

[Bar03] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[Bar04] C. Baral, K. Chancellor, N. Tran, N. Tran, A. Joy, and M. Berens. A knowledge based approach for representing and reasoning about signaling networks. In *Proceedings of ISMB/ECCB'04*, pp. 15–22. 2004.

[Bas] S. Baselice, P. Bonatti, and M. Gelfond. Towards an integration of answer set and constraint solving. In *Proceedings of ICLP'05*, pp. 52–66. Springer.

[Bie09] A. Biere, M. Heule, H. van Maaren, and T. Walsh (eds.). *Handbook of Satisfiability*. IOS Press, 2009.

[Bre96] G. Brewka and T. Eiter. Preferred answer sets for extended logic programs. In *Proceedings of KR'96*, pp. 86–97. Morgan Kaufmann Publishers, 1996.

[Dec03] R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

[Del] T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate functions in disjunctive logic programming: Semantics, complexity, and implementation in DLV. In *Proceedings of IJCAI'93*, pp. 847–852. Morgan Kaufmann Publishers.

[Dov09] A. Dovier, A. Formisano, and E. Pontelli. An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems. *Journal of Experimental and Theoretical Artificial Intelligence*, 21(2):79–121, 2009.

[Dre08] C. Drescher, M. Gebser, T. Grote, B. Kaufmann, A. König, M. Ostrowski, and T. Schaub. Conflict-driven disjunctive answer set solving. In *Proceedings of KR'08*, pp. 422–432. AAAI Press, 2008.

[Dre10a] C. Drescher, O. Tifrea, and T. Walsh. Symmetry-breaking answer set solving. In *Proceedings of ICLP'10 Workshop ASPOCP*. 2010. To appear.

[Dre10b] C. Drescher and T. Walsh. A translational approach to constraint answer set solving. In *Proceedings of ICLP'10*. Cambridge University Press, 2010. To appear.

[Eén06] N. Eén and N. Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.

[Eit99] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. The diagnosis frontend of the dlv system. *AI Communications*, 12(1-2):99–111, 1999.

[Eit08] T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12-13):1495–1539, 2008.

[Elk] I. Elkabani, E. Pontelli, and T. Son. Smodels with CLP and its applications: A simple and effective approach to aggregates in ASP. In *Proceedings of ICLP'04*, pp. 73–89. Springer.

[Erd] E. Erdem and M. Wong. Rectilinear Steiner tree construction using answer set programming. In *Proceedings of ICLP'04*, pp. 386–399. Springer.

[Geb07a] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. In *Proceedings of LPNMR'07*, pp. 260–265. Springer, 2007.

[Geb07b] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Proceedings of IJCAI'07*, pp. 386–392. AAAI Press/The MIT Press, 2007.

[Geb07c] M. Gebser, T. Schaub, and S. Thiele. GrinGo: A new grounder for answer set programming. In *Proceedings of LPNMR'07*, pp. 266–271. Springer, 2007.

[Geb09a] M. Gebser, H. Hinrichs, T. Schaub, and S. Thiele. xpanda: A (simple) preprocessor for adding multi-valued propositions to ASP. In *Proceedings of WLP'09*. 2009.

[Geb09b] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. On the implementation of weight constraint rules in conflict-driven ASP solvers. In *Proceedings of ICLP'09*, pp. 250–264. Springer, 2009.

[Geb09c] M. Gebser, M. Ostrowski, and T. Schaub. Constraint answer set solving. In *Proceedings of ICLP'09*, pp. 235–249. Springer, 2009.

[Gel88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of ICLP'88*, pp. 1070–1080. The MIT Press, 1988.

[Hel03] K. Heljanko and I. Niemelä. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming*, 3(4-5):519–550, 2003.

[Hua08] J. Huang. Universal booleanization of constraint models. In *Proceedings of CP'08*, pp. 144–158. Springer, 2008.

[Jaf94] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19-20:503–581, 1994.

[Jär09] M. Järvisalo, E. Oikarinen, T. Janhunen, and I. Niemelä. A module-based framework for multi-language constraint modeling. In *Proceedings of LPNMR'09*, pp. 155–169. Springer, 2009.

[Kat05] G. Katsirelos and F. Bacchus. Generalized nogoods in CSPs. In *Proceedings of AAAI'05*, pp. 390–396. AAAI Press, 2005.

[Lec96] M. Leconte. A bounds-based reduction scheme for constraints of difference. In *CP'96, Second International Workshop on Constraint-based Reasoning*. 1996.

[Leo05] N. Leone, G. Greco, G. Ianni, V. Lio, G. Terracina, T. Eiter, W. Faber, M. Fink, G. Gottlob, R. Rosati, D. Lembo, M. Lenzerini, M. Ruzzi, E. Kalka, B. Nowicki, and W. Staniszkis. The INFOMIX system for advanced integration of incomplete and inconsistent data. In *Proceedings of SIGMOD'05*, pp. 915–917. 2005.

[Lif02] V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1-2):39–54, 2002.

[Mel08a] V. Mellarkod and M. Gelfond. Integrating answer set reasoning with constraint solving techniques. In *Proceedings of FLOPS'08*, pp. 15–31. Springer, 2008.

[Mel08b] V. Mellarkod, M. Gelfond, and Y. Zhang. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence*, 53(1-4):251–287, 2008.

[Mit05] D. Mitchell. A SAT solver primer. *Bulletin of the European Association for Theoretical Computer Science*, 85:112–133, 2005.

[Nie99a] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.

[Nie99b] I. Niemelä, P. Simons, and T. Soininen. Stable model semantics of weight constraint rules. In *Proceedings of NMR'99*, pp. 317–333. Springer, 1999.

[Nie06] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[Nog01] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An A-prolog decision support system for the space shuttle. In *Proceedings of PADL'01*, pp. 169–183. Springer, 2001.

[Pal09] A. D. Palù, A. Dovier, E. Pontelli, and G. Rossi. Answer set programming with constraints using lazy grounding. In *Proceedings of ICLP'09*, pp. 115–129. Springer, 2009.

[Rei87] R. Reiter. Nonmonotonic reasoning. *Annual Review of Computer Science*, 2:147–187, 1987.

[Ros06] F. Rossi, P. van Beek, and T. Walsh (eds.). *Handbook of Constraint Programming*. Elsevier, 2006.

[Sim02] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[Sin05] C. Sinz. Towards an optimal cnf encoding of boolean cardinality constraints. In *Proceedings of CP'05*, pp. 827–831. Springer, 2005.

[Soi99] T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Proceedings of PADL'99*, pp. 305–319. Springer, 1999.

[Syr] T. Syrjänen. Lparse 1.0 user's manual.

[Tam06] N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear csp into sat. In *Proceedings of CP'06*, pp. 590–603. Springer, 2006.

[Wal00] T. Walsh. SAT v CSP. In *Proceedings of CP'00*, pp. 441–456. Springer, 2000.

[Wal06] T. Walsh. Symmetry breaking using value precedence. In *Proceedings of ECAI'06*, pp. 168–172. IOS Press, 2006.